

AD-A197 838

DTIC FILE COPY

2

7057-107

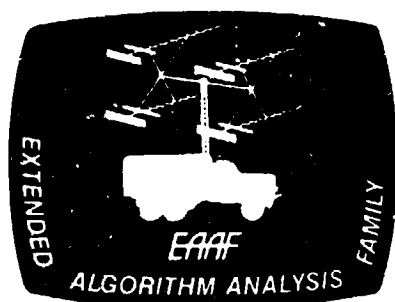
**U.S. ARMY INTELLIGENCE CENTER AND SCHOOL  
SOFTWARE ANALYSIS AND MANAGEMENT SYSTEM**

**APPLICATIONS OF PARALLELISM TO CURRENT  
ALGORITHMS FOR INTELLIGENCE ANALYSIS**

Martha Ann Griesel

J. Steven Hughes

Beth R. Moore



10 July 1987

National Aeronautics and  
Space Administration

**JPL**

**JET PROPULSION LABORATORY**  
California Institute of Technology  
Pasadena, California

JPL D-4719  
ALGO\_PUB\_0125

**DTIC**  
**ELECTE**  
JUL 27 1988  
**S D**

**DISTRIBUTION STATEMENT A**

Approved for public release;  
Distribution Unlimited

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ALGO PUB 0125	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  Applications of Parallelism to Current Algorithms for Intelligence Analysis		5. TYPE OF REPORT & PERIOD COVERED  FINAL
		6. PERFORMING ORG. REPORT NUMBER D-4719
7. AUTHOR(s) Dr. Ann Griesel, Steve Hughes, Beth Moore		8. CONTRACT OR GRANT NUMBER(s)  NAS7-918
9. PERFORMING ORGANIZATION NAME AND ADDRESS Jet Propulsion Laboratory, ATTN: 171-209 California Institute of Technology 4800 Oak Grove, Pasadena, CA 91109		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS  RE 182 AMEND #187
11. CONTROLLING OFFICE NAME AND ADDRESS Commander, USAICS ATTN: ATSI-CD-SF Ft. Huachuca, AZ 85613-7000		12. REPORT DATE 10 Jul 87
		13. NUMBER OF PAGES 21
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)  Commander, USAICS ATTN: ATSI-CD-SF Ft. Huachuca, AZ 85613-7000		15. SECURITY CLASS. (of this report)  UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE NONE
16. DISTRIBUTION STATEMENT (of this Report)  Approved for Public Dissemination		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES  Prepared by Jet Propulsion Laboratory for the US Army Intelli- gence Center and School's Combat Developer's Support Facility.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  Aggregation; Parallel Processors; Computer Architecture; Lock/Unlock; Monitors.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  For this study a spatial aggregation algorithm was implemented on two separate parallel processor machines - the JPL Mark II Hyper- cube and the ANL Sequent Balance. To remove test bias, the data set was pre-partitioned. The algorithm was successfully programmed and run on both machines. Technical details - number of pre- partitions, number of points per partition, program run time, comparison with sequential computers, etc. - are omitted.		

7057-107

U.S. ARMY INTELLIGENCE CENTER AND SCHOOL  
Software Analysis and Management System

Applications of Parallelism to Current Algorithms  
for Intelligence Analysis

10 July 1987


Authors:

  
Martha Ann Griesel

  
J. Steven Hughes


  
Beth R. Moore


Approval:

  
James W. Gillis, Subgroup Leader  
Algorithm Analysis Subgroup

  
Edward J. Records, Supervisor  
USAMS Task

Concur:

  
A. F. Ellman, Manager  
Ground Data Systems Section

  
Fred Vote, Manager  
Advanced Tactical Systems

JET PROPULSION LABORATORY  
California Institute of Technology  
Pasadena, California

JPL D-4719

## PREFACE

The work described in this publication was performed by the Jet Propulsion Laboratory, an operating division of the California Institute of Technology, under contract NAS7-918, RE182, A187 with the National Aeronautics and Space Administration, for the United States Army Intelligence Center and School.

This specific work was performed in accordance with the FY-87 statement of work (SOW #2).

## ACKNOWLEDGEMENT

The authors wish to acknowledge the valuable collaboration of John R. Gabriel, Argonne National Laboratory, in the development of this work.



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

## EXECUTIVE SUMMARY

This study examines implementation of a spatial aggregation algorithm on a hypercube and a shared memory machine with special attention given to data partitioning. The differences in implementation of the algorithm are due to data partitioning, data dependence, and communication between processors. Two parallel machines were used: The Cal Tech-JPL Mark II Hypercube, and the Sequent Balance at the Advanced Computing Research Facility at Argonne National Laboratory. The hypercube is a 32 node concurrent processor consisting of 32 identical processors linked by a communications network. The Sequent Balance is a high-performance, general-purpose computer system that uses 2 to 12 National Semi-conductor Series 32000 CPUs in a tightly-coupled multi-processing architecture. This study indicates that task oriented algorithms with a low degree of data interdependence are better suited to a shared memory implementation and data-driven algorithms to a hypercube implementation.

*Keywords: Parallel Processing,  
Spatial Aggregation, (1D)*

# CONTENTS

EXECUTIVE SUMMARY . . . . .	v
1. INTRODUCTION . . . . .	1
1.1 PURPOSE . . . . .	1
1.2 INITIAL PROPOSAL . . . . .	1
1.3 CONCURRENT PROCESSING APPLICATIONS . . . . .	1
2. ALGORITHM DESCRIPTIONS . . . . .	3
2.1 INTRODUCTION . . . . .	3
2.2 AGGREGATION ALGORITHM . . . . .	3
2.2.1 INTRODUCTION . . . . .	3
2.2.2 DESCRIPTION . . . . .	3
3. ARCHITECTURE DESCRIPTIONS . . . . .	5
3.1 INTRODUCTION . . . . .	5
3.2 HYPERCUBE . . . . .	5
3.3 SEQUENT BALANCE . . . . .	6
4. ALGORITHM IMPLEMENTATION . . . . .	7
4.1 INTRODUCTION . . . . .	7
4.2 HYPERCUBE IMPLEMENTATION . . . . .	7
4.2.1 INPUT/OUTPUT DESCRIPTION . . . . .	7
4.2.2 IMPLEMENTATION . . . . .	7
4.3 SEQUENT BALANCE IMPLEMENTATION . . . . .	8
4.3.1 MONITORS . . . . .	8
4.3.2 IMPLEMENTATION . . . . .	9
5. CONCLUSIONS . . . . .	10

**FIGURES**

Figures 1 and 2 . . . . . 11

Figures 3 and 4 . . . . . 12

**APPENDICES**

A. GLOSSARY . . . . . A-i

B. REFERENCES . . . . . B-1



## **1. INTRODUCTION**

### **1.1 PURPOSE**

The purpose of this document is to detail a study of concurrent processing by the Concurrent Processing Subgroup of the U.S. Army Intelligence Center and School (USAICS) Software Analysis and Management System (USAMS) task in association with the Advanced Computing Research Facility (ACRF) at Argonne National Laboratory (ANL). The study centered on the effect of different concurrent architectures (hypercube and shared memory) on Intelligence and Electronic Warfare (IEW) algorithm performance.

### **1.2 INITIAL PROPOSAL**

Increasing data rates and a growing diversity of IEW information sources create a void in current intelligence analysis methods. New ways are needed to quickly correlate and aggregate information being received from different types of sources (e.g. radar, imagery, and eyewitness reports).

The immediate need for intelligence analysis systems which process information more rapidly was addressed by examining applications of data distribution techniques to an existing aggregation algorithm. Since Jet Propulsion Laboratory (JPL) / USAMS has access to many of these unclassified algorithms, one was chosen and implemented on the Caltech/JPL Hypercube and VAX 11/780 (development machine) and an ACRF shared memory computer, the Sequent Balance.

### **1.3 CONCURRENT PROCESSING APPLICATIONS**

Parallel processing lends itself to certain types of applications. In order to exploit the use of many processors in an application, several issues must be addressed. One issue is identifying the dependencies between data structures and modules. If the degree of data dependence is large, the application is more suited to sequential processing than to parallel processing. A small number of data dependencies between modules indicates that the algorithm lends itself to parallel processing because modules do not have to rely on other modules and data structures for their information. [Ref. 3] Computations within each module and communication between modules proceed independently.

The best hardware for an application depends partly on the answer to this data dependency issue. This requires mapping the problem onto a variety of parallel architectures in order to find a good match for the specific application. [Ref. 3]

This report discusses mapping of a spatial aggregation algorithm onto hypercube and shared memory architectures. The algorithm was previously used in a "production" mode on sequential machines such as the CDC 6600, and on a vector processor (the CRAY I).

## 2. ALGORITHM DESCRIPTIONS

### 2.1 INTRODUCTION

Most algorithms that spend a substantial time in calculations can be modified to run faster and more efficiently on multiprocessors. If the data set is partitionable (able to be grouped into separate portions in a meaningful way), it is likely that parallel processing can speed up computation time. This is the aspect of algorithms that we are focusing on.

For each "candidate algorithm", the data set and its interaction with the data structures needs to be analyzed. If the data set can be meaningfully partitioned into smaller, distinct sets, clusters of associated data points can be sent to processing elements (PEs) and the data can be processed by the algorithm in parallel. If a section of the algorithm that spends a lot of time in computation can be isolated, then the PEs can each concurrently work on the computation using the available data sets. These are two of the areas of interest when implementing an algorithm on a parallel architecture.

### 2.2 AGGREGATION ALGORITHM

#### 2.2.1 INTRODUCTION

The algorithm that was chosen for this study was an aggregation algorithm. It uses a computationally intensive mathematical criterion to decide which points in two dimensional space belong together in a cluster and locates the "value center" of the cluster. The algorithm is used because:

- (1) it is used in real military simulation studies;
- (2) it performs a function equivalent to aggregating individual sighted objects (e.g. radios, trucks, helicopters); and
- (3) analysis on the uniprocessor for which it was originally designed showed that 70% of the CPU time is spent in one numeric calculation. This calculation is performed repeatedly, usually on independent data sets, making the algorithm a good candidate for parallel implementation.

#### 2.2.2 DESCRIPTION

The goal of the application program (using the aggregation algorithm) is to partition the input data points into separate clusters. For each individual cluster, a value center is

determined to maximize a given value function.

A pre-partitioned data set with distinct, disjoint groups of points was used for this initial implementation. The data partitions fall into distinct groups, thus clustering can be done in parallel. Each data set is sent to a PE point by point. When all points have been loaded on the PE, the clustering process begins. A cluster is a group of points that fall within an R-ball (radius-ball) of the center point. Each point is tried as a center point and a cluster is determined by using a precalculated radius. The next step is to determine what other points within the PE fall into the cluster. The number of clusters are minimized by checking for redundancy (small cluster contained in larger cluster) to determine which contain unique points. (See Figure 1)

The nonredundant clusters are fed to a simplex (polytope) maximization routine which uses a value function to calculate the value center of each cluster. Each PE, in turn will return a value center and its associated value for each cluster (See Figure 2). This part of the program is the most time consuming for a sequential processor.

### 3. ARCHITECTURE DESCRIPTIONS

#### 3.1 INTRODUCTION

The aggregation algorithm was implemented on two computers:

the Caltech/JPL Mark II Hypercube, and  
the Argonne National Laboratory Sequent Balance.

The Mark II Hypercube uses a "hypercube" architecture and the Sequent Balance uses a "shared memory" architecture. Each computer will be discussed separately.

#### 3.2 HYPERCUBE

The Caltech/JPL Mark II Hypercube (Cube) is a Multiple Instruction Multiple-Data (MIMD) machine which uses an N-cube interconnection network to link processing elements (nodes), each with its own Central Processing Unit (CPU) and memory.

Cubes come in configurations of  $2^n$  nodes. Each node is connected to  $n$  other nodes in an  $n$ -cube configuration and therefore communication between nodes requires at most  $n$  hops or the use of  $n-1$  intermediate nodes. The  $n$ -cube connection is realized by considering the binary representations of each node. Two nodes are connected if their binary representation differ by one bit. For example, given a 3-cube, or  $2^{**}3=8$  node cube, node 0 (000) is connected to node 1 (001), node 2 (010) and node 4 (100). To pass information from node 0 (000) to node 6 (110) would take two hops and use one intermediate node. Two possible paths that exist include the path of nodes (0 2 6) and (0 4 6).

The Mark II cube has 32 nodes. Each node consists of an 8086/8087 processor pair and 256K bytes of memory. The CPU speeds are .5 MIPS and .030 - .040 Mflops per node. Transfer rates between channels is about 8 Mb/s.

The cube that was currently available for general development was the Mark II cube on the Caltech Net (CITNET). This net includes the LOGOS VAX 11/750 connected via CITNET to Miranda, the intermediate host (IH) of the cube. LOGOS acts as the development machine, where developers code, compile, and test using a simulator, and ultimately download crosscompiled code to the cube via Miranda.

Two merits of a cube-connected architecture are as follows:

First, it is considered a good "in between" connection architecture. By this is meant that both the maximum distance between nodes and the number of connections per node scale as a function of  $\log(m)$ ,

where  $m$  is the number of nodes. In addition, the internal bandwidth of the machine increases faster than the number of nodes, the formula for connections being  $m \cdot \log(m)/2$ .

Second, mesh connections up to and including the dimensions of the cube can be realized allowing for easier parallel algorithm development.

### 3.3 SEQUENT BALANCE

The Balance 8000 is an expandable, high-performance, general purpose computer system that employs from 2 to 12 National Semiconductor Series 32000 CPU's in a tightly coupled multiprocessing architecture. The operating system, DYNIX, is a version of UNIX 4.2bsd that has been enhanced to support multiprocessing and to run on Series 32000 CPUs. [Ref. 5] The CPU's run at 10 MHz and may have their own local cache memory. Primary memory, which is shared by all CPU's, may be up to 26 Mbytes. The primary memory, all CPU's and all peripheral subsystems are connected via the SP8000 bus which has a maximum bandwidth of 26.67 Mbytes per second.

Being based on UNIX, multiprocessing is achieved by managing a queue of prioritized processes. The shared memory is a global resource which holds a single copy of the operating system, with the balance of the shared memory being assigned dynamically by page to individual processes. Scheduling of the processes for execution proceeds in a UNIX fashion, except that instead of a single CPU, a pool of CPU's is available for execution.

The CPU's are considered symmetric in that any processing in any state can execute on any available CPU. The system dynamically assigns processes to CPU's, balancing the load to take best advantage of each CPU. The shared memory concept allows an application developer to design an algorithm based on a queue of tasks to be performed. All tasks without interdependancies can then be executed in parallel, increasing throughput.

## **4. ALGORITHM IMPLEMENTATION**

### **4.1 INTRODUCTION**

Implementation of the aggregation algorithm was done on the two parallel architectures described above. Each implementation used the inherent features of this underlying architecture. In particular, the down-loading of data, data partitioning, and data dependency were different.

In working with the aggregation algorithm, it was discovered that the problem demanded a task-oriented approach. In such an approach, a larger problem is split up into smaller problems. Both architectures handled this approach well, but the Balance is architecturally more suited to task-oriented problems. The ordering of tasks among the PE's did not affect program efficiency. The differences in implementation of the algorithm are stressed in the following discussions.

### **4.2 HYPERCUBE IMPLEMENTATION**

#### **4.2.1 INPUT/OUTPUT DESCRIPTION**

Applying a problem to the cube involves verifying that the computational load for each node (or PE) is similar and that the time for internode communication required by the algorithm is minimized. Two programs are needed to run a problem on the cube. The element (ELT) program resembles a sequential computer program but invokes subroutine calls when it needs to communicate with other processors or with the host VAX via the Intermediate Host (IH) program. A copy of ELT executes on each node, whereas IH runs on the Intermediate Host machine.

Upon initializing the cube, each node is assigned a node number. These numbers are used by the IH and the other nodes for communication. When the IH communicates with the nodes or the nodes communicate with themselves, a minimum amount of information must be passed. A packet represents this information and is a set of 8 contiguous bytes.

When communication between the nodes or with the IH is desired, special commands for reading and writing are called. Other commands are available for communication between the nodes. [Ref. 6]

#### **4.2.2 IMPLEMENTATION**

Implementation of an algorithm on the cube requires that the database of points be downloaded onto the PE's. Since the database is pre-partitioned, there is no problem with load

balancing. When there are an unknown number of points being downloaded to the PE's and there is no control over how many points each of the PE's get, then load balancing becomes an issue. Downloading the data is accomplished with single-stream execution. Once all data is loaded, all processors work in parallel on their clusters using the aggregation algorithm. When finished, each PE sends back their results to the IH, which completes the job by reporting the results, again by single-stream execution. (See Figure 3)

Data is distributed evenly to the PE's, one point at a time. One of the disadvantages of this method is the inability to download a contiguous block of data to a single PE. This problem was solved for this initial study by partitioning the data in advance.

Each PE has a copy of the program to work on along with its data cluster. It is important to note that while each PE is executing in parallel (working on its cluster set), individual clusters contained in the cluster sets are also being worked on.

After the computations are complete, the PE's contain the value centers of their assigned clusters. In order to send back the points so that they will be interpreted correctly by the IH, it is important that the data be "packaged" in the correct format. Sending the address of the buffer containing the results to the IH and allowing enough buffer space to receive all information from each of the PE's is crucial for receiving results on the IH.

#### 4.3 SEQUENT BALANCE IMPLEMENTATION

##### 4.3.1 MONITORS

There are several solutions to problems inherent in a parallel processing environment.

"One solution that is efficiently implementable in every parallel environment is the concept of monitors. A monitor is a conceptual abstraction composed of three distinct parts: (1) the data that are shared, (2) the operations that represent critical sections (sections of code that can safely be executed by only a single process at a time) associated with the shared data, and (3) the code that is required to initialize the shared structures. Only one process may be "in a monitor" at a given time. This is achieved conceptually by setting a lock at the point where the operation is entered and releasing the lock at the point where the operation is exited. The fundamental property of monitors is to



ensure exclusive access to resources and to manage synchronization requirements (waiting and signaling). The higher-level synchronization primitives (locks, ASKFOR monitor (see definitions)) are implemented as macros that invoke lower-level macros. A macro package is available that uses macros to implement the basic operations for each machine at the ACRF and for each language. This provides a portable implementation of a few of the high-level synchronization patterns yielding less complex code and smaller chance of error." [Ref. 7 and 8]

#### 4.3.2 IMPLEMENTATION

Our implementation utilizes the ASKFOR monitor. This requires each PE to "ask for" the next available task to perform. The monitor manages the pool of outstanding tasks. [Ref. 8] The task at hand is to take a set of points (a cluster set) and to group them into clusters. The value center of the cluster is determined and written to a data structure in shared memory. The data structure update is controlled by the lock/unlock synchronization primitive (this ensures mutual exclusion). The monitor gives each PE a task to complete. When one PE completes its task, it receives another task to be performed. Each PE uses the nonlinear optimization routine and the value function to determine the value center for its cluster. (See Figure 4) The downloading of data and setting up the data structures is performed with single-stream execution. Processing of the data in the computational part of the program is done in parallel.

The Balance downloads data in contiguous blocks to the PE's. This allows a partitioned data set to be efficiently distributed amongst the PE's. Since the Balance uses shared memory, access to shared resources is a crucial issue. In order for each PE to receive an "unique" cluster set, it is necessary to use the synchronization primitive of lock/unlock to ensure exclusive access to the shared data. The locking mechanism is also used when the PE's write the value centers of their clusters to the result array shared by all PE's.

## 5. CONCLUSIONS

There are many possible ways to adapt existing sequential algorithms to concurrent architectures. The approach taken depends on both the type of algorithm and the available architecture. Despite differences in approaches, there exist similarities in problems that lend themselves to parallel processing. Data that is partitionable into distinct, meaningful groups and computation intensive algorithms are two attributes of a problem which allow it to efficiently use concurrency.

The development of the data structures for the partitioned data set and the downloading of the partitioned data to the PE's was easier and more straightforward in this initial pre-partitioned case on the Sequent Balance shared memory machine than on the cube. The use of monitors on the Balance and the accessibility of memory eased implementation. I/O was awkward with the cube architecture, but some of those problems are rectified in the new Mark III Hypercube.

Since the Balance relies on shared memory, data dependence is an important issue. The accessibility of resources and data structures is controlled by monitors and locks. This is not necessary with the cube. Separate data structures for each processing element are necessary for parallel processing of the algorithm on the Hypercube. Thus, this implementation indicated that task oriented algorithms are more suited for the Sequent Balance as long as the data has a low degree of interdependence, and the Hypercube is better suited for algorithms that lend themselves to data driven implementations. Clearly, with readily partitionable data, shifting the intensive computation to concurrently processing PE's is more efficient than sequential processing. Data partitioning is the key for both implementations. Since partitioning is done sequentially in a preprocessing phase, little is gained by either implementation if partitioning is "too difficult". In particular, this would be the case if the data partitioning problem is equivalent in computational complexity to the non-linear optimizations performed in the PE's. You have nearly backed the problem up to its dual. The next step of this research is to study the mapping of this dual problem into the architectures.

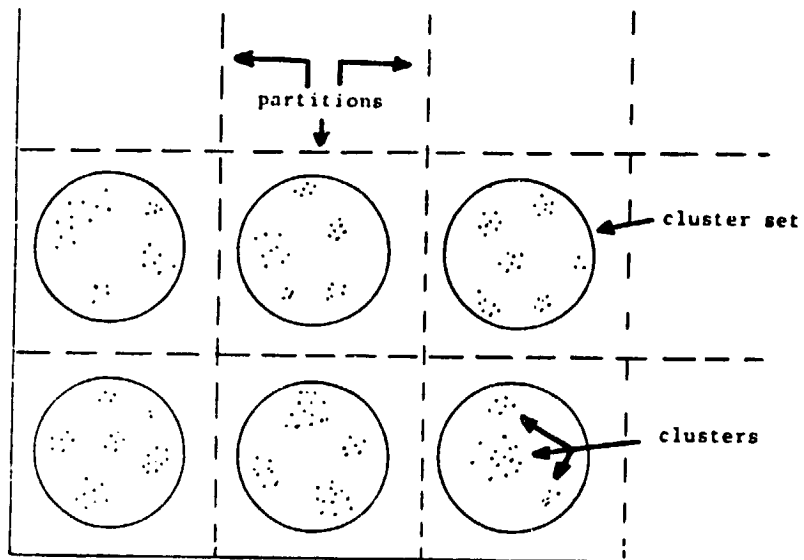


Figure 1. Pre-partitioned data set containing cluster sets

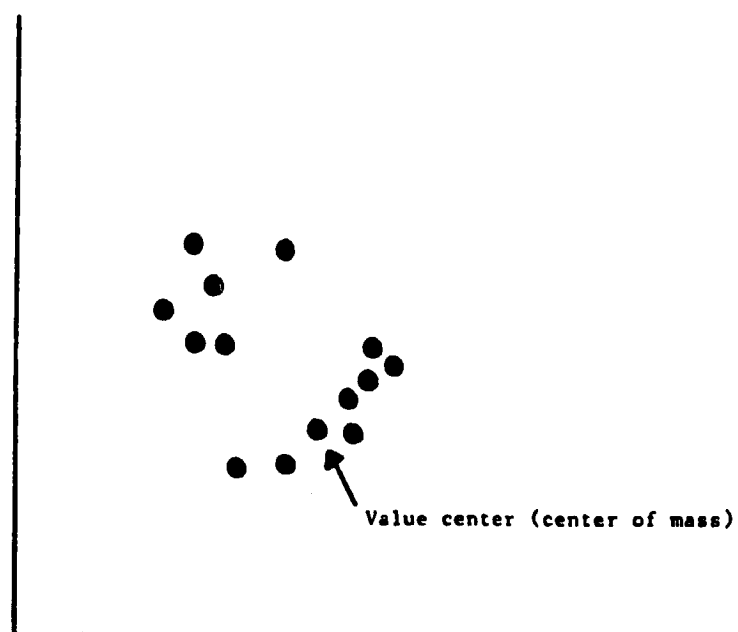


Figure 2. Value Center for one cluster

## Hypercube Implementation

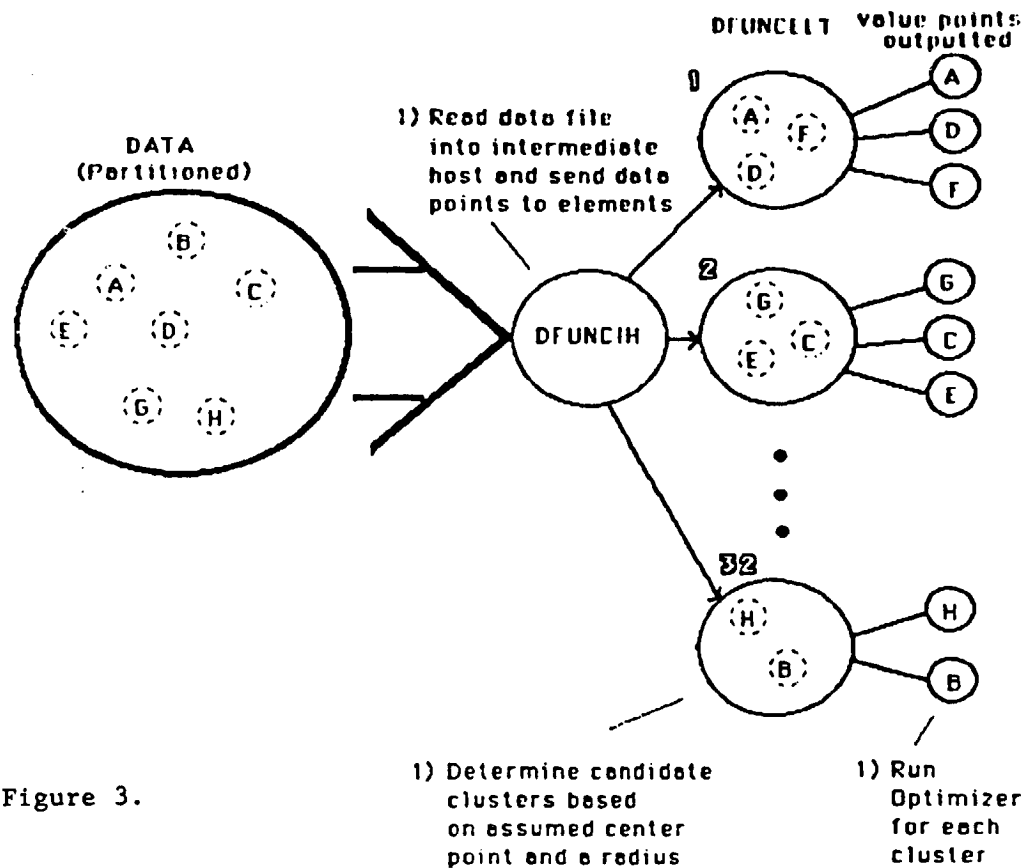


Figure 3.

## Shared Memory Implementation

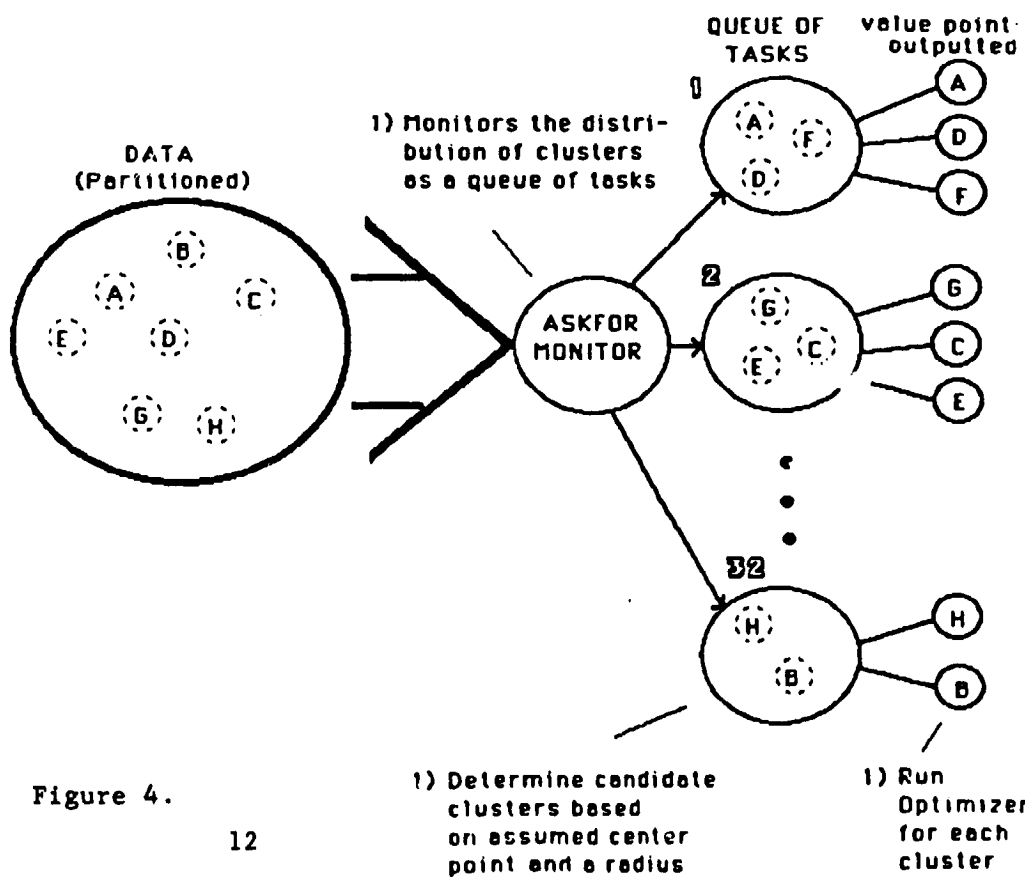


Figure 4.

## APPENDIX A

### GLOSSARY

The following is a list of terms as used in this report.

**Process** - synonymous with task. It is an autonomic unit of activity in a computer system.

**Sequential Process** - an entity that executes a series of tasks on a uni-processor (a computer with one central processing unit). This entity usually consists of a data structure and a sequential program that operates on it. [Ref. 1] Within the sequential program, statements are executed one by one. The results of program execution with the same data set is always the same; there is no program speed up as long as the operations are carried out in the same sequence. [Ref. 1] This is not true of a concurrent process.

**Concurrent Process** - more than one process working on a single problem at the same time. Concurrent Processing includes both Parallel Processing and Distributed Processing.

**Multiprocessing** - executing more than one process by switching back and forth between the processes.

**Parallel Processing** - several processes executing at the same time on one or more different problems. This usually happens on a multiprocessor architecture, where several autonomous processors can each execute separate programs.

**Shared Memory** - an area of memory that is used by more than one processor.

**Data Dependency** - when data produced by one part of a concurrent process is used by another part. [Ref. 2]

**Algorithm** - a precise method usable by a computer for the solution of a problem. It is composed of a finite set of steps, each of which may require one or more operations. [Ref. 4]

**Cluster** - a term that refers to a group of points that fall within an r-ball (radius-ball) of the center point of the group.

**Cluster Set** - a set of clusters.

**Lock/Unlock** - Provide mutual exclusion among processes (a shared memory high-level synchronization pattern).

**Askfor Monitor** - Coordinate a pool of processes working on a pool of subproblems (a shared memory high-level synchronization pattern).

**Uni-Processor Architecture** - a computer with a single central processing unit (CPU).

**Multiprocessor Architecture** - a computer with a number of processing elements (PE's).

**Data Driven** - Each CPU is dependent on the data that is fed to it from other CPU's. Tasks are assigned to CPU's before the algorithm is executed.

**Task Oriented** - The operating system dispatchs tasks as the CPU's become available.

## APPENDIX B

### REFERENCES

1. Hansen, Per Brinch, "The Architecture of Concurrent Programs", Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.
2. "DEC Professional", Professional Press, Inc., Spring House, PA, 1987.
3. Moldovan, Dan I., "Modern Parallel Processing", University of Southern California, 1986.  
(Manuscript due to be published)
4. Horowitz, Ellis and Sahni, Sartaj, "Fundamentals of Computer Algorithms", Computer Science Press, Inc., 1978.
5. Sequent Computer Systems, Inc., "Balance 8000 System Technical Summary", 1984.
6. Enguehard, S., "A Beginner's Guide to Programming the Caltech Hypercube, Volume 1", Hm 120, Dec. 1984.
7. Lusk, Ewing, (Reference from a talk on Parallel Programming Methodology at Jet Propulsion Laboratory), Advanced Computing Research Facility, Argonne National Laboratory, 1986.
8. Lusk, E. L., Stevens, R. L., Overbeek, R. A., "A Tutorial on the Use of Monitors in C: Writing Portable Code for Multiprocessors", Advanced Computing Research Facility, Argonne National Laboratory.